JNIF

Java Native Instrumentation Framework

Luis Mastrangelo & Matthias Hauswirth
luis.mastrangelo@usi.ch
matthias.hauswirth@usi.ch

Faculty of Informatics
Università della Svizzera italiana

September 25, 2014
PPPJ'2014
Cracow University of Technology

Introduction

- Program analysis tools are important in many software engineering tasks

Introduction

- Program analysis tools are important in many software engineering tasks
    - Profiling

Introduction

- Program analysis tools are important in many software engineering tasks
  - Profiling
  - Debugging

Introduction

- Program analysis tools are important in many software engineering tasks
  - Profiling
  - Debugging
  - Program optimization

Introduction

- Program analysis tools are important in many software engineering tasks
  - Profiling
  - Debugging
  - Program optimization
- Dynamic program analysis is one the main approaches, generally based on instrumentation, i.e.

Introduction

- Program analysis tools are important in many software engineering tasks
  - Profiling
  - Debugging
  - Program optimization
- Dynamic program analysis is one the main approaches, generally based on instrumentation, i.e.
  - The ability to add or change instructions to a target program in order to observe a desired property.

Background

- Within the JVM, instrumentation can be done with a custom classloader, java agent or JVMTI.

Background

- Within the JVM, instrumentation can be done with a custom classloader, java agent or JVMTI.
- A java agent is loaded in the same VM as the target application and after the bootstrap phase, thus increasing the perturbation of the target application and all events in the bootstrap phase are missing.

Background

- Within the JVM, instrumentation can be done with a custom classloader, java agent or JVMTI.
- A java agent is loaded in the same VM as the target application and after the bootstrap phase, thus increasing the perturbation of the target application and all events in the bootstrap phase are missing.
- JVMTI agents are written in native code (usually C or C++).

Background

- Within the JVM, instrumentation can be done with a custom classloader, java agent or JVMTI.
- A java agent is loaded in the same VM as the target application and after the bootstrap phase, thus increasing the perturbation of the target application and all events in the bootstrap phase are missing.
- JVMTI agents are written in native code (usually C or C++).
- Instrumentation using JVMTI are more accurate because it can control the entire JVM, catching all event in the bootstrap phase.

Background

- Within the JVM, instrumentation can be done with a custom classloader, java agent or JVMTI.
- A java agent is loaded in the same VM as the target application and after the bootstrap phase, thus increasing the perturbation of the target application and all events in the the bootstrap phase are missing.
- JVMTI agents are written in native code (usually C or C++).
- Instrumentation using JVMTI are more accurate because it can control the entire JVM, catching all event in the bootstrap phase.
- But usually requires an external process to implement the instrumentation itself.

Use Custom ClassLoader

- Requires to modify target application.

Use Custom ClassLoader

- Requires to modify target application.
- Some applications already use a custom classloader for their purposes.

Java instrument package

- Run inside the JVM

Java instrument package

- Run inside the JVM
  - Special attention to avoid instrumenting the instrumentation itself.

Java instrument package

- Run inside the JVM
  - Special attention to avoid instrumenting the instrumentation itself.
- Activates after the JDK have been loaded

Java instrument package

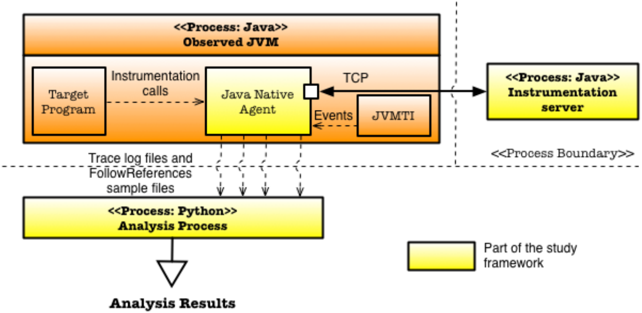
- Run inside the JVM
  - Special attention to avoid instrumenting the instrumentation itself.
- Activates after the JDK have been loaded
  - Unable to instrument JDK

Java Native Agent

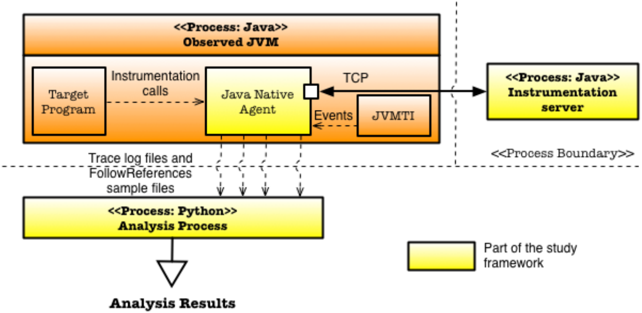
Java Native Agent implemented with JVMTI API.

- Ask the instrumentation server to instrument all classes in the JVM.
- Hooks on every event of interest
    - Class creation
    - Instrumentation calls

Motivation: External process

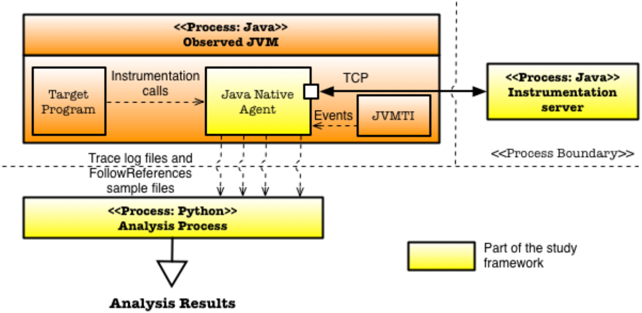


Motivation: External process



- A Java Native Agent attached to the observed program.

Motivation: External process



- A Java Native Agent attached to the observed program.
- An Instrumentation Server for bytecode instrumentation.

Motivation

- Instrument bytecode using JVMTI native agents.

Motivation

- Instrument bytecode using JVMTI native agents.
- C++ library/framework that allows instrumentation and analysis of Java bytecode.

JNIF

Main Goal

Instrument and analize Java bytecode using C++

JNIF

Main Goal

Instrument and analize Java bytecode using C++

- Ability to parse and write java class files.

JNIF

Main Goal

Instrument and analize Java bytecode using C++

- Ability to parse and write java class files.
- Object model to query every item in a class file.

JNIF

Main Goal

Instrument and analize Java bytecode using C++

- Ability to parse and write java class files.
- Object model to query every item in a class file.
- Stack map frames generation.

JNIF

Main Goal

Instrument and analize Java bytecode using C++

- Ability to parse and write java class files.
- Object model to query every item in a class file.
- Stack map frames generation.
  - Control flow graph representation.
  - Data flow equation to type check every method.

JNIF

Main Goal

Instrument and analize Java bytecode using C++

- Ability to parse and write java class files.
- Object model to query every item in a class file.
- Stack map frames generation.
    - Control flow graph representation.
    - Data flow equation to type check every method.
- Parse and write returns the original bytecode (nothing get change): Important property to make test cases.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.
- Join points in the control flow graph presents an important issue

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.
- Join points in the control flow graph presents an important issue
  - To compute the join type between two classes the class hierarchy is needed.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.
- Join points in the control flow graph presents an important issue
  - To compute the join type between two classes the class hierarchy is needed.
  - This makes the analysis non-local.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.
- Join points in the control flow graph presents an important issue
  - To compute the join type between two classes the class hierarchy is needed.
  - This makes the analysis non-local.
  - Imposes knowing the whole class hierarchy.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.
- Join points in the control flow graph presents an important issue
    - To compute the join type between two classes the class hierarchy is needed.
    - This makes the analysis non-local.
    - Imposes knowing the whole class hierarchy.
    - Class loading issues.

JNIF: Type checking and stack map frames

- Abstract interpretation of every JVM instruction.
- To compute stack and local variables types.
- Accurately model JVM type hierarchy.
- Join points in the control flow graph presents an important issue
  - To compute the join type between two classes the class hierarchy is needed.
  - This makes the analysis non-local.
  - Imposes knowing the whole class hierarchy.
  - Class loading issues.
  - The class path must be replicated in the instrumentation server in order to search for the common super class.

JNIF: Reading a class file

```
const char* data = ...;
int len = ...;

jnif::ClassFile cf(data, len);
```

JNIF: Reading & Writing

```
const char* data = ...;
int len = ...;
jnif::ClassFile cf(data, len); // Parse buffer

// Analyze or edit the ClassFile ...

// Encode the ClassFile into binary
int newlen = cf.computeSize();
u1* newdata = new u1[newlen];
cf.write(newdata, newlen);

// Use newdata and newlen ...

delete [] newdata; // Free the new binary
```

JNIF: Traversing methods

```
const char* data = ...;
int len = ...;
jnif::ClassFile cf(data, len);

for (jnif::Method* m : cf.methods) {
  cout << "Method:␣";
  cout << cf.getUtf8(m->nameIndex);
  cout << cf.getUtf8(m->descIndex);
  cout << endl;
}
```

JNIF: Instrumenting constructor

```
ConstIndex mid = cf.addMethodRef(classIndex,
    "alloc", "(Ljava/lang/Object;)V");

for (Method* method : cf.methods) {
  if (method->isInit()) {
    InstList& instList = method->instList();

    Inst* p = *instList.begin();
    instList.addZero(OPCODE_aload_0, p);
    instList.addInvoke(OPCODE_invokestatic, mid,p);
  }
}
```

```
InputStream is;
if (args.length == 0) {
    is = new FileInputStream("");
} else {
    is = new ByteArrayInputStream(null);
}
// What is the type of is at this point?
```

Evaluation

- Dacapo benchmarks to evaluate performance
- Compare compute frames in JNIF and ASM
- In-process JNIF vs out-of-process ASM
- Metrics
  - Overhead in instrumentation time
  - Parser, writer, compute frames
  - Total running time

ASM Instrumentation server

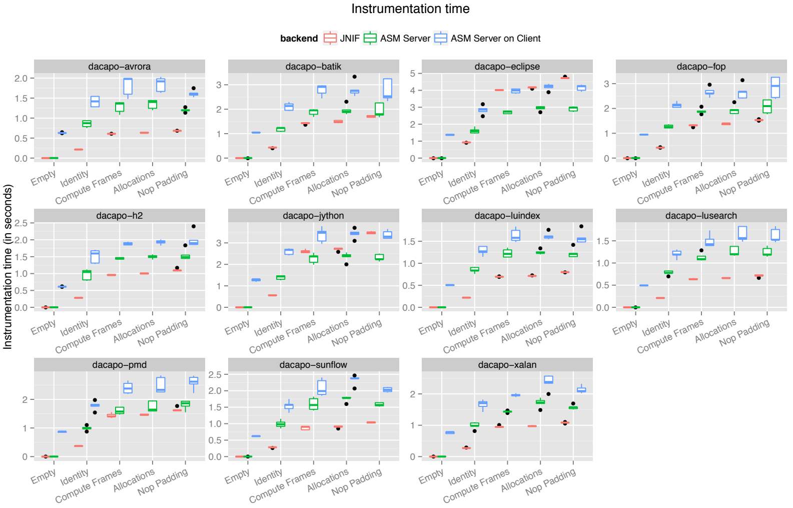
Main Goal

Instrument every class that is requested

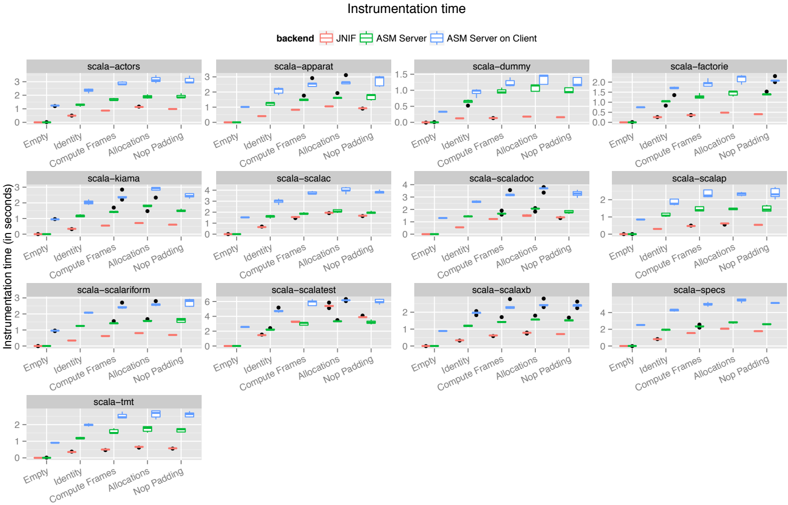
- Receives instrumentation TCP requests with class file bytecodes.
- Uses ASM [1] for instrumenting the class files.
- Responses with instrumented class files.
- The instrumented class files invokes native methods on a predefined class that are implemented by the Java Native Agent.

[1] http://asm.ow2.org/

Evaluation: DaCapo Instrumentation time



Evaluation: Scalabench Instrumentation time



Limitations

- Class files that contains JSR/RET instructions are not supported

Limitations

- Class files that contains JSR/RET instructions are not supported
    - Class files requiring stack maps do not include JSR/RET.

Limitations

- Class files that contains JSR/RET instructions are not supported
    - Class files requiring stack maps do not include JSR/RET.
    - ASM does the same.

Limitations

- Class files that contains JSR/RET instructions are not supported
    - Class files requiring stack maps do not include JSR/RET.
    - ASM does the same.
- Partial support for invokedynamic

Limitations

- Class files that contains JSR/RET instructions are not supported
  - Class files requiring stack maps do not include JSR/RET.
  - ASM does the same.
- Partial support for invokedynamic
  - Initial successful tests with JRuby.

Limitations

- Class files that contains JSR/RET instructions are not supported
    - Class files requiring stack maps do not include JSR/RET.
    - ASM does the same.
- Partial support for invokedynamic
    - Initial successful tests with JRuby.
- Improve data-flow algorithm

Limitations

- Class files that contains JSR/RET instructions are not supported
  - Class files requiring stack maps do not include JSR/RET.
  - ASM does the same.
- Partial support for invokedynamic
  - Initial successful tests with JRuby.
- Improve data-flow algorithm
  - Issues when bytecode has several exception handler entries.

Thanks!

Suggestions/Questions/Feedback